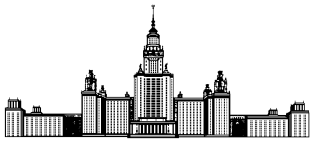
Московский Государственный Университет им. М.В. Ломоносова



Факультет Вычислительной Математики и Кибернетики
Кафедра Математических Методов Прогнозирования

Курсовая работа

«Регуляризация тематических моделей в библиотеке BigARTM»

Выполнил:
студент 3 курса 317 группы
Апишев Мурат Азаматович

Научный руководитель:
д.ф-м.н., доцент
Воронцов Константин Вячеславович.

Москва, 2014

Содержание

1 Введение

Тематическое моделирование — активно развивающаяся в последние годы область машинного обучения. Оно позволяет решать задачи тематического поиска, категоризации и кластеризации корпусов текстовых документов. Аналогичные задачи решаются для коллекций изображений и видеозаписей.

Тематическая модель определяет, к каким темам относится каждый документ, а также то, какие термины из словаря образуют ту или иную тему.

Одной из проблем существующих алгоритмов обучения является невозможность совмещения большого числа функциональных требований в одной модели. LDA, который является на сегодняшний день стандартом в данной области, не позволяет комбинировать более 2-3 требований из-за сложности используемого математического аппарата.

В работе [1] рассматривается теория аддитивной регуляризации тематических моделей (APTM), построенная над более простым алгоритмом обучения PLSA и позволяющая решить описанную проблему. Библиотека BigARTM представляет собой программную реализацию этой концепции, адаптированную под мультипроцессорное и кластерное распараллеливание.

Список основных задач данной курсовой работы: произвести обзор BigARTM, реализовать в ней механизм добавления/удаления регуляризаторов, описать возможности его использования и модификации.

Работа имеет следующую структуру: в разделе 2 введены базовые обозначения и определения, необходимые для дальнейшего изложения; алгоритм обучения тематических моделей и идея регуляризации вводятся в разделе 3; в разделе 4 приводится краткий обзор существующих параллельных библиотек тематического моделирования; раздел 5 описывает общую архитектуру библиотеки и схему пользовательского взаимодействия с BigARTM; раздел 6 посвящён реализации и использованию регуляризаторов в библиотеке; в 7 разделе показаны эксперименты с регуляризаторами; раздел 8 предназначен для выводов и подведения итогов курсовой работы.

2 Терминология

Прежде всего рассмотрим некоторые базовые понятия и необходимые обозначения.

Вероятностная тематическая модель (BTM) описывает каждую тему дискретным распределением на множестве терминов, каждый документ — дискретным распределением на множестве тем. Предполагается, что коллекция документов — это последовательность терминов, выбранных случайно и независимо из смеси таких распределений, и ставится задача восстановления компонент смеси по выборке.

- D — коллекция текстовых документов.
- W — словарь коллекции текстов.
- T — множество тем.

Документы в коллекции можно представить в виде так называемого «мешка слов». В

рамках этой концепции документ рассматривается как множество терминов из словаря и соответствующих им счётчиков частот встречаемости.

При рассмотрении коллекции в виде пар (d, w), где w — номер термина, а d — номер документа, вводятся следующие счётчики частот:

- n_{dw} — число вхождений термина w в документ d;
- $n_d = \sum_{w \in W} n_{dw}$ — длина документа d в терминах;
- $n_w = \sum_{d \in D} n_{dw}$ — число вхожденией документа w во все документы коллекции;
- $n = \sum_{d \in D} \sum_{w \in d} n_{dw}$ — длина коллекции D в терминах;

Если же рассматривать коллекцию в виде троек (d, w, t), где d, w и t — номера соответствующих документа, термина и темы, то можно ввести такие счётчики:

- n_{dwt} — число троек, в которых термин w встретился в документе d и связан с темой t;
- $n_{dt} = \sum_{w \in W} n_{dwt}$ — число троек, в которых термин из документа d связан с темой t;
- $n_{wt} = \sum_{d \in D} n_{dwt}$ — число троек, в которых термин w связан с темой t;
- $n_t = \sum_{d \in D} \sum_{w \in d} n_{dwt}$ — число троек, связанных с темой t;

С использованием данных счётчиков можно ввести следующие частотные оценки вероятностей, связанных со скрытой переменной t:

- $\hat{p}(w|t) = \dfrac{n_{wt}}{n_t}, \quad \hat{p}(t|d) = \dfrac{n_{dt}}{n_d},$

Вводятся следующие матрицы:

- $\Phi = (\phi_{wt})_{W \times T}$, $\phi_{wt} = \hat{p}(w|t)$ — матрица «термины-темы».
- $\Theta = (\theta_{td})_{T \times D}$, $\theta_{td} = \hat{p}(t|d)$ — матрица «темы-документы».
- F — заданная матрица частот «термины-документы», $F \approx \Phi\Theta$.

Задача тематического моделирования — найти матрицы Φ и Θ, максимизирующие следующий функционал:

$$L(\Phi, \Theta) = \sum_{d \in D} \sum_{w \in d} n_{dw} \sum_{t \in T} \phi_{wt} \theta_{td} \to \max_{\Phi, \Theta} \tag{1}$$

3 Обучение тематических моделей

3.1 PLSA

Вероятностный латентный семантический анализ (PLSA) был предложен Т.Хофманном в [2].

Принимается гипотеза условной независимости, утверждающая, что вероятность появления термина в данном документе зависит только от темы этого термина и не зависит от документа. Вероятностная порождающая модель PLSA имеет следующий вид:

$$p(w|d) = \sum_{t \in T} p(w|t)p(t|d) \tag{2}$$

PLSA можно реализовать с помощью EM-алгоритма. Итерационный процесс состоит из двух шагов — E-шага (Expectation) и M-шага (Maximization). На E-шаге по текущим значениям ϕ_{wt} и θ_{td} с помощью формулы Байеса вычисляются условные вероятности $p(t|d, w)$:

$$H_{dwt} = p(t|d, w) = \frac{\phi_{wt}\theta_{td}}{\sum_{s \in T} \phi_{ws}\theta_{sd}}$$

На M-шаге по условным вероятностям H_{dwt} вычисляются новые приближения параметров ϕ_{wt} и θ_{td}. Используются указанные в предыдущем разделе формулы:

$$\phi_{wt} = \frac{n_{wt}}{n_t}, \quad \theta_{td} = \frac{n_{dt}}{n_d},$$

Однако данная версия алгоритма плохо подходит для задачи параллельной обработки бальших массивов данных по двум причинам:

- За время одного прохода по коллекции распределения терминов в темах (матрица Ф) успевают много раз сойтись, а распределния тем в документах проходят лишь одну итерацию.
- Трёхмерная вспогательная матрица H_{dwt} становится чрезвычайно большой.

Избавиться от этих проблем позволяет т.н. пакетный онлайновый EM-алгоритм. Пакетным он называется потому, что вся коллекция делится на блоки документов, обрабатываемых независимо. Онлайновость означает потоковую обработку коллекции, когда модель дообучается, получая на вход всё новые документы.

Помимо описанных преимуществ, данная модификация позволят также не хранить в явном виде матрицу Θ. Вероятности θ_{td} не нужны по окончании обработки документа d, поэтому матрицу $(\theta_{td})_{T \times D}$ можно заменить вектором $(\theta_t)_T$.

Замечание: Работа онлайнового алгоритма отличается для больших и малых коллекций. Для большой коллекции достаточно одного прохода по всем документам, в то время как маленькие требуют многократного прохода. В случаях малых и динамических коллекций (т.е.меняющих со временм свою тематику) значимость пакетов убывает по мере поступления новых, поэтому необходимо ввести параметр $\rho_j \in (0, 1]$, отвечающий за скорость «забывания» старых оценок:

$$\hat{n}_{wt} := \rho_j \hat{n}_{wt} + \tilde{n}_{wt}$$

Алгоритм Online Batch PLSA реализует все описанные идеи. Он выглядит следующим образом:

ЗДЕСЬ БУДЕТ АЛГОРИТМ ИЗ voron2013ptm. ToDo

Такой алгоритм хорошо параллелится. Θ выводится на нодах, Φ — на мастере.

3.2 Аддитивная регуляризация

Неоднозначность матричного разложения $F \approx \Theta\Phi$ даёт свободу выбора матриц из правой части равенства, позволяя наложить на тематическую модель дополнительные требования. Модифицируем максимизируемый функционал 1:

$$L(\Phi, \Theta) + R(\Phi, \Theta) \to \max_{\Phi,\Theta} \tag{3}$$

$$R(\Phi, \Theta) = \sum_{i=1}^{n} \tau_i R_i(\Phi, \Theta)$$

где $R_i(\Phi, \Theta)$ — дополнительные требования к модели, τ_i — неотрицательные коэффициенты регуляризации, выполнены условия неотрицательности и нормировки столбцов матриц Φ и Θ.

Решение этой задачи приводит к обощению формул M-шага в EM-алгоритме:

$$\phi_{wt} = \frac{\left(n_{wt} + \phi_{wt}\dfrac{\partial R}{\partial \phi_{wt}}(\Phi, \Theta)\right)_+}{\sum_{u \in W}\left(n_{ut} + \phi_{ut}\dfrac{\partial R}{\partial \phi_{ut}}(\Phi, \Theta)\right)_+}, \quad \theta_{td} = \frac{\left(n_{dt} + \theta_{td}\dfrac{\partial R}{\partial \theta_{td}}(\Phi, \Theta)\right)_+}{\sum_{s \in T}\left(n_{ds} + \theta_{sd}\dfrac{\partial R}{\partial \theta_{sd}}(\Phi, \Theta)\right)_+} \tag{4}$$

n_{wt} и n_{dt} определяются аналогично из формул предыдущего раздела.

Таким образом, суть добавления регуляризаторов — в простом изменении формул M-шага.

Замечание: Использование регуляризаторов требует аккуратного выстраивания т.н. траектории регуляризации. Этот процесс включает в себя настройку параметров, определение времени подключения/отключения того или иного регуляризатора, используемого в модели и т.п.

3.3 Регуляризатор Дирихле

Рассмотрим популярную на сегодняшний день модель обучения LDA (latent Dirichlet allocation). Она основана на разложении 2 при дополнительном предположении, что векторы документов $\theta_d = (\theta_{td} \in \mathbf{R}^{|T|})$ и векторы тем $\phi_t = (\phi_{wt} \in \mathbf{R}^{|W|})$ порождаются распределениями Дирихле с параметрами $\alpha \in \mathbf{R}^{|T|}$ и $\beta \in \mathbf{R}^{|W|}$ соответственно. В работе [1] показано, что отличие LDA от PLSA — в сглаживании ϕ_{wt} и θ_{td}:

$$\phi_{wt} = \frac{\hat{n}_{wt} + \beta_w}{\hat{n}_t + \beta_0}, \quad \theta_{td} = \frac{\hat{n}_{td} + \alpha_t}{\hat{n}_d + \alpha_0}, \quad \alpha_0 = \sum_t \alpha_t, \ \beta_0 = \sum_w \beta_w \tag{5}$$

Т.е. LDA — это PLSA со встроенным регуляризатором сглаживания (здесь и далее — регуляризатор Дирихле). Сглаживание может быть полезным для некоторых столбцов матриц, но в общем случае оно противоречит тому, что матрицы Φ и Θ должны быть сильно разреженными.

4 Существующие реализации параллельных алгоритмов

В данном разделе обзорно рассматриваются несколько существующих популярные реализаций параллельных алгоритмов тематического моделирования. Все они основаны на модели LDA, имеют разные технические и алгоритмические детали. Рассмотрим их основные преимущества и недостатки, после чего опишем

4.1 Краткий обзор

AD-LDA Данный алгоритм был предложен в работе [4]. В AD-LDA был реализован только межпроцессорный параллелизм, возможность работы на кластере не обсуждалась. Данная работа является одной из первых, в которых была осознана возможность параллелизации процеса обучения модели, и в этом её серьёзное достоинство. С другой стороны, в технической реализации имеется серьёзные недостаток — собранные на разных процессорах статистики сливаются вместе во время специально выделяемого шага синхронизации. Это приводит к тому, что скорость работы алгоритма определяется скростью работы самого медленного процессора. Оставляет желать лучшего отказоустойчивость системы. Нагрузка на сеть так же неравномерна — во время синхронизации она чрезвычайно высока, в прочие же временные промежутки сеть простаивает. Кроме того, в данном алгоритме каждый процессор обладает своей матрицей Φ, что приводит к неоправданным расходам памяти.

PLDA PLDA был описан в статье [7]. Алгоритм представляет собой модификацию AD-LDA, в которой параллелизм реализован в двух вариантах: с помощью MPI и Map-Reduce (при этом работа опять же распределяется между процессорами, кластерная реализация отсутствует). Авторам удалось добиться повышения отказоустойчивости, однако сделано это было путём увеличения потребления дисковой памяти (которая и в AD-LDA потреблялась в слишком большом объёме). Основной недостаток AD-LDA — шаг синхронизации — так же был унаследован PLDA.

Y!LDA Следующим шагом для параллельных алгоритмов обучения тематических стала работа Y!LDA, описанная в [5]. Авторы описали недостатки Ad-LDA и предложили свой вариант параллелизма, теперь уже как в рамках одной машины, так и в кластере. Основным достижением стало избавление от выделенного шага синхронизации. Авторы отметили его основные недостатки (плохая отказоустойчивость, неравномерная нагрузка на сеть и ожидание медленных процессоров) и предложили в рамках одной машины ввести специальный поток синхронизации, к которому ядра будут обращаться независимо друг от друга. Кроме того, было принято решение хранить на одной машине только одну копию матрицы Φ, общую для всех процессоров.

Для кластера авторы предложили использовать т.н. «архитектуру классной доски». Суть её состоит в том, что каждый компьютер в сети обращается к глобальной матрице Ф независимо от других, обновляя нужные статистики. Порцией обновления при этом является один термин. Кроме того, было предложено и хорошее техническое решение — модель (матрица Ф) хранится в `memcached`. Кластерный параллелизм был реализован в Hadoop. [1].

PLDA+ В этой публикации была предпринята попытка модифицировать PLDA. Авторы сделали алгоритм параллельным в кластерном смысле. Для того, чтобы побороть проблемы, связанные с шагом синхронизации, было решено использовать конвейерную архитектуру. Для этого всё множество процессоров делится на две группы — вычислители и коммутаторы. Первая группа производит сэмплирование (собственно расчёт модели), вторая отвечает за своевременную доставку данных. Вкупе с равномерным распределением коллекции в виде блоков по процессорам-вычислителям, а также наличием механизма приоритетов, сглаживающем узкие места во время счёта, это даёт, согласно экспериментам, хорошие результаты. Тем не менее, один существенный недостаток у данной реализации имеется — значимая часть вычислительных ресурсов тратится не на счёт, а на своевременную доставку данных.

4.2 BigARTM

Из всего вышеописанного было сделано несколько выводов:

1. Алгоритм обязательно должен быть асинхронным.
2. Матрица Ф должна быть локальной в рамках компьютера, а не ядра.
3. Использование `memcached` является хорошей идеей [2] .
4. В качестве кластера лучше использовать MPI, а не Hadoop.
5. Коллекцию следует разделить на блоки, обрабатываемые независимо.

Все эти идеи положены в основу BigARTM. Они либо уже реализованы, либо будут реализованы к релизу библиотеки.

5 Библиотека BigARTM

BigARTM — библиотека тематического моделирования, реализующая концепцию аддитивной регуляризации. На данный момент библиотека поддерживает многопроцессорный паралеллизм. Основным алгоритмом является описанный в 3.1 Online Batch PLSA.

[1] memcached — связующее программное обеспечение, реализующее сервис кэширования данных в оперативной памяти на основе парадигмы хеш-таблицы.

[2] К сожалению, сервис memcached предназначен для использования только под Linux. Тем не менее, это не является большой проблемой, поскольку большинство кластеров работает именно под Unix-подобными ОС и кластерный параллелизм BigARTM будет ориентирован именно под них. Windows-версия библиотеки, по сути, предназначена для использования на локальной машине. Хотя, возможно, к релизу для Windows-версии будет написан сервис, осуществляющий функции memcached, как это уже сделано в рамках многопроцессорного параллелизма.

Protocol buffers [3] Рассмотрим кратко эту технологию, поскольку в BigARTM она используется очень интенсивно. Protocol Buffers позволяет описывать proto-сообщения на псевдоязыке, которые затем можно преобразовать специальным компилятором (protoc) в структуры данных со всеми необходимыми методами на C++, Python и Java. Ключевой особенностью является возможность обмена сообщениями между этими языками программирования посредством механизма serializer/deserializer, переводящего сообщения в байт-массив и обратно. Данное решение является максимально переносимым и унифицированным.

Все конфигурации библиотеки, такие как настройки тематической модели, параметры регуляризаторов и т.п., определяются и передаются только посредством proto-сообщений. Результирующая модель также описывается соответствующим сообщением.

Замечание: Все необходимые для работы с библиотекой сообщения будут описаны при дальнейшем изложении.

5.1 Краткое описание

Представление коллекции Коллекция представляется в виде «мешка слов» (Bag of words). Поскольку библиотека предназначена для обработки больших массивов текстовой информации, она является онлайновой. Механизм потоковой загрузки и обработки информации реализуется разделением коллекции на пакеты (`Batch`), которые обрабатываются по очереди. Каждый `Batch` имеет свой словарь, который может быть как локальным (что предпочтительнее), так и совпадающим со всем словарём коллекции.

Приведём здесь соответствующее proto-сообщение:

```
message Batch {
  repeated string token = 1;
  repeated Item item = 2;
}
```

Первое поле — набор терминов (словарь), второе — набор документов (см. 5.1).

Замечание: В силу внутренних особенностей работы BigARTM, оптимальное число `Batches` — в 4-5 раз больше, чем число используемых процессоров. Этот параметр не повлиет на качество результирующей модели, но скажется на производительности.

Представление документов В BigARTM реализована гибкая концепция представления данных. Каждый документ является экземпляром класса `Item`. Этот объект, помимо самого документа (хранимого в виде последовательности терминов и их счётчиков), `Item` может содержать произвольные метаданные, связанные с ним. К таким данным относятся информация об авторе, дате публикации, ссылках на документ и из него, тегах и т.п. Всё это может оказаться крайне полезным при использовании тех или иных регуляризаторов.

Рассмотрим proto-сообщение для `Item`

```
message Item {
  optional int32 id = 1;
```

[3] Подробнее о том, что такое Google Protocol Buffers и как с ними работать, можно прочесть в [3]

```
  repeated Field field = 2;
}
```

Здесь `id` — идентификатор документа, второе поле — набор всевозможных объектов в тексте. Объектом может быть сам текст документа, строка с авторами, список тегов и т.п. Работа механизма становится ясной, если взглянуть на определение сообщения для `Field`

```
message Field {
  optional string field_name = 1 [default = "@body"];
  repeated int32 token_id = 2;
  repeated int32 token_count = 3;
}
```

Первое поле сообщает о том, что это за объект (по-умолчанию — основной текст документа). Второе поле содержит список идентификаторов терминов, встречающихся в этом тексте. В третьем находятся соответствующие этим терминам счётчики встречаемости в этом документе.

Входные и выходные данные На вход функциям библиотеки могут подаваться как исходные тексты, так и коллекция в виде пакетов. Это крайне удачное решение, так как, во-первых, коллекция в пакетном виде занимает более чем в два раза меньше памяти [4] , чем в исходном текстовом, во-вторых, библиотеке не придётся производить процедуру преобразования, что сэкономит время. На выходе пользователь получает собственно тематическую модель (матрицу Φ).

Особенности архитектуры Рассмотрим архитектуру библиотеки на концептуальном уровне. Для каждого участвующего в работе алгоритма процессора создаётся соответствующий ему объект класса `Processor`. Процессоры взаимодействуют независимо друг от друга. Задача процессора — производить собственно вычисления. У процессорного элемента имеется своя копия матрицы Φ, которую он использует. Также у каждого `Processor` имеется т.н. очередь процессора, в которую поступают пакеты данных. Процессор анализирует их, выводя матрицу Θ (сама матрица при этом явным образом в памяти не хранится). После полученные результаты (счётчики статистик) отправляются в виде блоков в т.н. очередь слияния. Очередь слияния — это место, откуда получает информацию экземпляр класса `Merger`. Этот объект отвечает за объединение результатов, полученных от всех процессоров, в одно целое и обновление матрицы Φ. Эта матрица используется в работе процессоров. При этом хранится две копии матрицы Φ — новая и старая. Старая матрица является доступной для чтения всем процессорам, новая доступна в режиме read-write только `Merger`. В неё записываются обновления статистик, полученные во время очередного прохода по коллекции. После окончания прохода старая матрица замещается новой (операция дешёвая, происходит простое копирование указателей), а под следующую новую матрицу выделяется память.

Ключевые отличия и заимствования Обращаясь к разделу 4, сравним кратко архитектуру BigARTM с существующими реализациями. Некоторые идеи, использованные

[4] Никакой архивации при этом не производится. Каждый Batch представляет собой набор Item в виде сериализованных proto-сообщений. Это было сделано из соображений удобства использования, а объём уменьшается в виде побочного эффекта.

при создании BigARTM, были позаимствованы из ранних работ по схожей тематике ([?], [4]). сравнения не было, но будет, как только получим кластерное рапараллеливание

Далее будут рассмотрены основные шаги по настройке и использованию существующей версии BigARTM, а также список нововведений, которые будут в неё внесены в будущем (либо уже внесены, но требуют доработки).

Замечание: На данный момент единственным поддерживаемым библиотекой языком является Python, все выкладки будут производится на нём.

Замечание: Здесь и далее под внешней итерацией работы алгоритма понимается один проход по всей коллекции, а под внутренней — один проход по одному документу (`Item`).

5.2 Установка

Для установки библиотеки необходимо выполнить следующую последовательность действий:

1. Установить и распаковать boost 1.55, после чего присвоить системной переменной `BOOST\_ROOT` путь к корню распакованной папки (если переменная не существует, необходимо создать её).
2. Скачать по адресу `https://s3-eu-west-1.amazonaws.com/artm/libs_win32_v110.7z` статические библиотеки, необходимые для работы, и распаковать в папку `libs` в корневой директории BigARTM.
3. Скачать по адресу `http://miru.hk/archive/ZeroMQ-4.0.3~miru1.0-x86.exe` библиотеку ZeroMQ, распаковать её, и присвоить системной переменной `ZEROMQ32\_ROOT` путь к корневой директории ZeroMQ (если переменная не существует, необходимо создать её).
4. Установить Python 2.7 (x32)
5. Добавить корневую папку Python в переменную окружения `PATH`.
6. Следуйте инструкциям, описанным в файле

   `BigARTM_ROOT_DIRECTORY\3rdparty\protobuf\python\README.txt`

5.3 Работа с библиотекой

Запуск обучения Основным классом, обеспечивающем пользовательский API, является `MasterComponent`. Объект данного класса содержит в себе все созданные тематические модели и регуляризаторы, через него производится управление процессом обучения.

Для того, чтобы запустить обучение тематической модели, требуется пошаговое выполнение следующей инструкции:

1. Ядро библиотеки представляет собой скомпилированный модуль `artm.dll`. Прежде всего требуется подключить его. Чтобы это сделать, необходимо поместить в программу следующие строки:

```
address = os.path.abspath(os.path.join(os.curdir, os.pardir))
os.environ['PATH'] = ';'.join([address +
  '\\Win32\\Release', os.environ['PATH']])
library = ArtmLibrary(address + '\\Win32\\Release\\artm.dll')
```

2. Необходимо создать объект класса `MasterComponent`. Для этого нужно описать соответствующее proto-сообщение, которое (в базовой конфигурации) требует заполнения следующих полей:

```
optional int32 processors_count = 2 [default = 1];
optional string disk_path = 3;
```

Первое поле представляет собой число процессоров, в рамках которых будет производится распараллеливание алгоритма. Второе поле описывает путь к папке, в которую будут сохранятся данные, преобразованные в `Batches` (кроме того, именно в этой папке библиотека будет искать пакеты для своей работы). Создание и описание этого сообщения может иметь следующий вид:

```
master_config = messages_pb2.MasterComponentConfig()
master_config.processors_count = 1
master_config.disk_path = 'disk_path'
```

После того, как конфигурационное сообщение сформировано, можно создать сам объект `MasterComponent`:

```
master_component = library.CreateMasterComponent(master_config)
```

3. Следующим шагом требуется загрузить коллекцию и словарь для неё. Данное действие является техническим, в качестве базового примера можно использовать считывание, описанное в `python_client.py`.

4. Теперь в `master_component` нужно добавить тематические модели для обучения. Допустим, что требуется обучить одну модель. Для её создания необходимо заполнить соответствующее proto-сообщение. Оно будет выглядеть так:

```
optional string name = 1 [default = ""];
optional int32 topics_count = 2 [default = 32];
optional int32 inner_iterations_count = 4 [default = 10];
repeated Score score = 7;
```

Параметры имеют следующие назначения: `name` — имя тематической модели; `topic_counts` — число тем, которые будет искать в коллекции данная модель; третий параметр назначает модели число внутренних итераций; в последнем поле указываются функционалы качества, которые наобходимо рассчитывать для данной модели в ходе её работы [5] . Создать конфигурацию модели можно так:

```
model_config = messages_pb2.ModelConfig()
model_config.topics_count = 20
model_config.inner_iterations_count = 10
```

[5]На данный момент в библиотеке реализована только перплексия.

```
score_ = model_config.score.add()
score_.type = 0
```

`score_.type = 0` соответствует добавлению модель требование подсчёта перплексии на каждой итерации. Теперь можно создать саму модель, отнеся её к созданному ранее объекту `MasterComponent`, после чего активировать:

```
model = master_component.CreateModel(master_component, model_config)
```

5. После того, как модель была создана, необходимо запустить работу алгоритма [6] . Для этого нужно описать цикл по числу внешних итераций (это число определяется пользователем), внутри которого должны быть такие строки кода:

   ```
   master_component.InvokeIteration(1)
   master_component.WaitIdle();
   ```

   Первая строка производит вызов одной внешней итерации работы алгоритма, вторая — ожидание завершения выполнения это итерации.

   Этого достаточно для запуска алгоритма, однако в большинстве случаев требуется контролировать его работу (в т.ч. и следить за перплексией на данной итерации). Для этого следует выгрузить посчитанную на данный момент модель и просмотреть её параметры. Выгрузить модель можно, добавив ещё одну строку под предыдущими:

   ```
   topic_model = master_component.GetTopicModel(model)
   ```

   Как было указано ранее, сама модель описывается proto-сообщением. Это сообщение имеет такой вид:

   ```
   optional string name = 1 [default = ""];
   optional int32 topics_count = 2;
   optional int32 items_processed = 3;
   repeated string token = 4;
   repeated FloatArray token_weights = 5;
   optional DoubleArray scores = 6;
   ```

   Первые два поля были описаны ранее и имеют тот же смысл. Третье содержит число обработанных на данный момент документов (учитываются все проходы по каждому документу, включая повторные). В четвёртом поле лежит словарь для данной модели. В пятом — веса каждого термина, полученные в результате работы алгоритма (матрица Ф). Последнее поле содержит значения счётчиков на данной итерации (в описываемом примере — значение перплексии). Данную информацию можно извлекать после каждой итерации и использовать для оценивания качества обучения.

   Выгруженная после последней внешней итерации тематическая модель является финальным результатом работы алгоритма.

6. В случае необходимости перенастройки `master_component` или `model` используется функция `Reconfigure()`:

[6] Вообще говоря, прежде, чем начинать счёт, в модель стоит добавить регуляризаторы. О том, как это сделать, подробно написано в соответствующем разделе.

```
master_component.Reconfigure(new_master_config)
model.Reconfigure(new_model_config)
```

7. После окончания работы модели удалять её не нужно. Все модели будут автоматически удалены при вызове деструктора `MasterComponent`.

Замечание: Базовых типов `DoubleArray` и `FloatArray` в protocol buffers нет, это пользовательские тип, эквивалентные вещественному массиву (работать с ними нужно по тем же правилам, что и остальными сообщениями — через создаваемый protocol buffers интерфейс).

Замечание: Все функции типа `Reconfigure()` являются не до конца тестированными, поэтому должны использоваться с осторожностью.

Замечание: Пример пользовательского приложения над BigARTM можно найти в файле `BigARTM\_ROOT\_DIRECTORY\src\python_client\python_client`. Выдержки кода, указанные в вышеописанной инструкции, заимствованы оттуда.

5.4 Планируемые нововведения

Ниже приведён список различных модификаций, которые появятся в release-версии библиотеки:

- Кластерный параллелизм.
- Возможность работы в Linux.
- Интерфейсы на C++ и Java.
- Коллекция регуляризаторов.
- Поддержка 64-битной архитектуры.
- Усовершенствованный механизм хранения данных, необходимых для работы регуляризаторов.
- Коллекция функционалов качества.
- Хранение матриц Φ и Θ в разреженном виде.

6 Регуляризаторы в BigARTM

Ключевым отличием BigARTM от других библиотек алгоритмов тематического моделирования является наличие регуляризаторов. Задача данного раздела — описать API, предоставленный библиотекой для работы с существующими реализациями регуляризаторов, а также пояснить схему добавления новых.

6.1 Кракое описание механизма регуляризации в BigARTM

Поскольку части матрицы Θ вычисляются независимо на разных процессорах, регуляризация Θ производится в каждом процессоре на всех внутренних итерациях. Объектом регуляризации является столбец матрицы (т.е. документ). Управлять процессом регуляризации Θ можно только между внешними итерациями. С матрицей Φ ситуация иная. Регуляризация происходит в $Merger$, после того, как все процессоры на данной внешней итерации отработали, и новая матрица Φ сформирована. Регуляризация в логическом смысле опять работает со столбцом (т.е. с темой), но технически регуляризаторы вызываются для каждого элемента матрицы. При этом нормировочная константа, необходимая для выполнения М-шага 3 алгоритма, автоматически поправляется. Φ регуляризуется только на тех внешних итерациях, на которых это указал пользователь вызовом соответствующей функции (см. ниже).

6.2 Существующие регуляризаторов

Все регуляризаторы описываются своими proto-сообщениями, содержащими необходимые данные и параметры. Конфигурационное сообщение для регуляризатора матрицы Θ должно содержать по одному набору параметров для каждой внутренней итерации. Сообщение для регуляризатора матрицы Φ наполняется одним набором параметров для одной внешней итерации. По истечении внешней итерации все конфигурации могут быть обновлены.

На данный момент в BigARTM реализовано по одному базовому регуляризатору для каждой из матриц Φ и Θ, информация о которых приведена ниже:

Имя регуляризатора	**protobuf-сообщение**

Регуляризатор сглаживания/разреживания для Θ	`message SmoothSparseThetaConfig {`
	`  repeated double alpha_0 = 1;`
	`  repeated DoubleArray tilde_alpha = 2;`
	`}`

Регуляризатор сглаживания/разреживания для Φ	`message SmoothSparsePhiConfig {`
	`  required int32 background_topics_count = 1;`
	`  required double beta_0 = 2;`
	`  required DoubleArray tilde_beta = 3;`
	`  repeated double background_beta_0 = 4;`
	`  repeated DoubleArray background_tilde_beta = 5;`
	`}`

Смысл регуляризаторов сглаживания/разреживания и их параметров Регуляризатор сглаживания/разреживания [7] представляет собой модификацию сглаживающего регуляризатора Дирихле (встроенного в модель LDA и описанного в разделе 3). Несмотря

[7] Подробное описание и лингвистические обоснования этого регуляризатора можно найти в [10]

на простоту, уже данный регуляризатор решает задачи более сложные, чем регуляризатор Дирихле.

Разделим множество тем T на предметные S и фоновые B. Зададим вектор-столбцы параметров таким образом, чтобы в обеих матрицах Φ и Θ темы из S одинаково разреживались, а темы из B — сглаживались каждая собственным образом. Это приводит к тому, что

- каждая тема из S приобретает некоторое выраженное ядро терминов, отличающее её от остальных тем;
- каждая фоновая тема из B сглаживается, причём вариация параметров приводит к тому, что разные темы выполняют различные задачи (выделение стоп-слов, общей лексики коллекции и т.п.).

Замечание: В данной реализации регуляризатора фоновыми считаются $|B|$ тем с конца.

Для регуляризатора Θ каждый набор параметров состоит из вектора длиной в число тем (`tilde_alpha`) и коэффициента при этом векторе (`alpha_0`). После поэлементного перемножения данный вектор должен содержать отрицательные числа для тем из S, и положительные — для тем из B.

Регуляризатор для Φ требует в явном виде указать число фоновых тем (поле `background_topics_count`) — это нужно для проверки корректности параметров. `tilde_beta` — вектор длиной с текущее число терминов в матрице Φ, предназначенный для разреживания тем из S. `beta_0` — коэффициент при нём. Последние два параметра имеют тот же смысл и предназначены для фоновых тем, по одному набору на каждую.

6.3 Подключение регуляризаторов

Теперь, на примере `SmoothSparseTheta`, будет рассмотрен поэтапно процесс включения регуляризатора в тематическую модель. Предполагается, что переменная `topic_counts` содержит общее число тем, выделяемых данной моделью, а `background_topics_count` — число тем, объявленных фоновыми.

1. Прежде всего следует описать конфигурационное сообщение для регуляризатора. Оно может быть описано так:

```
regularizer_config_theta = messages_pb2.SmoothSparseThetaConfig()
for i in range(0, inner_iterations_count):
  regularizer_config_theta.alpha_0.append(1)
  tilde_alpha_ref = regularizer_config_theta.tilde_alpha.add()
  for j in range(0, topics_count - background_topics_count):
    tilde_alpha_ref.value.append(-1)
  for j in range(0, background_topics_count):
    tilde_alpha_ref.value.append(1)
```

2. Любой регуляризатор в BigARTM используется только в сообщении-оболочке `Regularizer`, являющемся общим для всех регуляризаторов и имеющем вид:

```
message RegularizerConfig {
  enum Type { ... }

  required string name = 1;
  required Type type = 2;
  required bytes config = 3;
}
```

В данном сообщении первое поле — имя добавляемого регуляризатора, второе — тип (описанным выше регуляризаторам сглаживания/разреживания соответствуют типы 2 и 3), третье поле — описанная выше конфигурация самого регулризатора, сериализованная функциями protocol buffers в массив байтов. В рассматриваемом примере возможно следующее определение этой оболочки:

```
general_regularizer_config_theta = messages_pb2.RegularizerConfig()
general_regularizer_config_theta.name = 'regularizer_theta'
general_regularizer_config_theta.type = 2
general_regularizer_config_theta.config =
  regularizer_config_theta.SerializeToString()
```

3. После описания сообщения-оболочки, можно создать объект регуляризатора:

   ```
   regularizer_theta = master_component.CreateRegularizer(
     general_regularizer_config_theta)
   ```

   Добавляемый регуляризатор сохранён в данном `MasterComponent`, все модели, ассоциированные с этим `MasterComponent`, могут использовать его.

4. Теперь требуется сообщить нужной тематической модели о том, что при её обучении должен использоваться `regularizer_theta`. Для этого требуется модификации шага 4 в алгоритме работы с библиотекой. Перед созданием модели её конфигурация дополняется ещё одной строкой:

   ```
   model_config.regularizer_name.append('regularizer_theta')
   ```

   Таким образом модель «узнаёт» имя регуляризатора, который нужно будет использовать. Разные модели могут иметь разные списки используемых регуляризаторов. При этом все эти регуляризаторы, как уже было отмечено, будут храниться в активном экземпляре `MasterComponent`.

5. В случае, когда на данной внешней итерации нужно произвести регуляризацию Φ, необходимо в шаге 5 алгоритма использования библиотеки после строки

   ```
   topic_model = master_component.GetTopicModel(model)
   ```

   сразу вставить строку

   ```
   model.InvokePhiRegularizers();
   ```

   Выгрузку модели нужно произвести раньше, поскольку её данные могут быть нужны для регуляризации. В случае, если данная внешняя итерация является последней,

для учёта регуляризации требуется ещё раз выгрузить модель уже после вызова регуляризации и полученный результат считать финальным.

Стоит обратить внимание на то, что при регуляризации будут вызваны все регуляризаторы матрицы Φ, ассоциированные с данной моделью.

6. Часто возникает потребность заменить параметры существующего регуляризатора. Для этого необходимо описать новое конфигурационное сообщение, которое, после сериализации, нужно присвоить полю `config` в сообщении-обёртке `general_regularizer_config_theta`. После этого вызывается функция перенастройки:

   ```
   regularizer_theta.Reconfigure(general_regularizer_config_theta)
   ```

7. Для удаления регуляризатора из модели достаточно удалить его имя из списка `model_config.regularizer_name` и реконфигурировать модель обновлённой `model_config`. Сам `regularizer` будет удалён автоматически при завершении работы `MasterComponent`.

6.4 Создание нового регуляризатора

[8]

Далее описан процесс создания нового регуляризатора и добавления его в библиотеку. Данные действия подчиняются некоторым общим правилам, поэтому будут изложены в алгоритмическом виде:

1. В первую очередь требуется описать в файле `messages.proto` protobuf-сообщение — конфигурацию нового регуляризатора, после чего добавить соответствующий элемент в поле `Type` в сообщении-оболочке `RegularizerConfig`.

2. Затем необходимо написать `.h` и `.cc` файлы создаваемого регуляризатора и добавить их в проект `libartm`. В данном пункте следует учитывать несколько деталей, которые будут рассмотрены подробнее в конце данного раздела.

3. После нужно добавить в файл `c_interface.cc` `#include` с именем `.h` файла нового регуляризатора.

4. В `c_interface.cc` надо найти функцию `ArtmReconfigureRegularizer()`. В ней есть switch по типам регуляризаторов, в который требуется добавить case с типом своего регуляризатора (для этого можно просто скопировать один из существующих case и поменять в нём имена регуляризатора и его конфигурации).

5. Для того, чтобы использовать добавленный регуляризатор, осталось скомпилировать `messages.proto` (используя компилятор `protoc`) в файлы на C++ и Python, после чего перекомпилировать проекты `libartm` и `artm`.

[8] Данный раздел не имеет отношения к пользовательской документации и предназначен для разработчиков BigARTM. Описанная в нём информация требует более глубокого понимания устройства библиотеки

Рекомендации по написанию файлов регуляризаторов. Любой регуляризатор, описанный в библиотеке, должен удовлетворять нескольким требованиям:

- Регуляризатор обязательно пишется (как и всё ядро библиотеки) на C++ (.cc и .h файлы).
- Регуляризатор представляет собой класс-наследник класса `RegularizerInterface`. Данный класс содержит два метода

  ```
  bool RegularizeTheta(const Item& item,
      std::vector<float>* n_dt, int topic_size, int inner_iter)
  bool RegularizePhi(TopicModel* topic_model)
  ```

  Как понятно из названия, каждый метод отвечает за регуляризацию соответствующей матрицы.
- Класс регуляризатора вместе со всем своим содержимым должен быть описан в namespace `regularizer`.
- Технически допустима реализация в одном классе регуляризатора сразу для обеих матриц, однако рекомендуется этого избегать и описывать один (в математическом смысле) регуляризатор в двух разных классах и работать с ними как с разными объектами. Это существенно упростит конфигурацию отдельного регуляризатора, использование его данных в функциях регуляризации, а также избавит от необходимости задавать ненужные данные, когда потребуется регуляризация только одной матрицы.
- Файл `.h` будет отличаться для разных регуляризаторов только названием, достаточно скопировать какой-нибудь существующий (регуляризирующий ту же матрицу) и поменять необходимые имена типов.
- Файл `.cc` будет включать в себя реализацию соответствующего метода регуляризации. Сама реализация, в свою очередь, состоит из трёх концептуальных этапов — считывания данных из объекта-конфигурации, проверки корректности этих данных и собственно регуляризации. Считывание данных рекомендуется производить в структуры тех же типов, что и используемые в конфигурации — т.е. описанные в файлах `messages.pb.h` и `messages.pb.cc` protobuf-типы. Во время проверки корректности ввода следует возвращать из функции `false` в случае выявления несоответствия. Наконец, после окончания этапа регуляризации следует вернуть `true`.

Замечание: В целом, при написании нового регуляризатора рекомендуется опираться на существующие реализации, это может сэкономить немало времени.

7 Эксперименты

7.1 Описание экспериментов

В данном разделе будет рассмотрен модельный эксперимент с использованием регуляризаторов в BigARTM. Его задачей является демонстрация использования реализованного

механизма регуляризаторов, от него не ожидается получения ответов на какие-либо исследовательские вопросы.

7.1.1 Описание эксперимента

Производится обучение двух тематических моделей — с регуляризатором и без. Для регуляризации модели используется регуляризатор сглаживания/разреживания. Параметры эксперимента следующие:

1. Обучающая коллекция документов — NIPS ($\approx$ 1600 документов).
2. Объём словаря — $\approx$ 13000 терминов.
3. Число внешних итераций — 12, внутренних — 10.
4. Число процессоров — 2.
5. Общее число тем — 18, фоновых тем — 3 (для модели без регуляризатора все темы равнозначны).

Регуляризация имеет следующую траекторию:

- 1 — 4 итерации:
- 5 — 8 итерации:
- 9 — 12 итерации:

ToDO

7.1.2 Результаты эксперимента

Сравнение качества моделей производилось по перплексии на обучающей выборке. Соответствующие графики построены на рис ... Из них видно, что регуляризация влияет на конечный результат. То, что это влияние не столь существенно — вина неаккуратной настройки параметров модели и регуляризатора (они неоптимизированные, не учитывают текущего состояния модели). Кроме того, перплексия на обучающей выборке — не лучший функционал качества. Существенно более важны такие параметры, как различность тем, разреженность матриц Φ и Θ (без учёта фоновых тем), объём ядер тем (т.е. слов, характеризующих данную тему). Регуляризация имеет перед собой задачу оптимизировать именно эти величины, не ухудшив при этом переплексию по сравнению с нерегуляризованной моделью.

8 Заключение

ToDo

Список литературы

[1] Воронцов К.В. Вероятностное тематическое моделирование, 2013.

[2] T. Hofmann Probabilistic latent semantic indexing // Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval. — New York, NY, USA: ACM, 1999. — Pp. 50–57.

[3] https://developers.google.com/protocol-buffers/docs/tutorials

[4] David Newman, Arthur Asuncion, Padhraic Smyth and Max Welling — Distributed Algorithms for Topic Models, 2009.

[5] A. J. Smola and S. Narayanamurthy — An architecture for parallel topic models, 2010.

[6] Arthur Asuncion, Padhraic Smyth, Max Welling — Asynchronous Distributed Estimation of Topic Models for Document Analysis, 2010.

[7] Yi Wang, Hongjie Bai, Matt Stanton, Wen-Yen Chen and Edward Y.Chang — PLDA: Parallel Latent Dirichlet Allocation for Large-Scale Applications, 2009.

[8] Ke Zhai, Jordan Boyd-Graber, Nima Asadi, Mohamad Alkhouja — Mr. LDA: A Flexible Large Scale Topic Modeling Package using Variational Inference in MapReduce, 2012.

[9] Liu, Z., Zhang, Y., Chang, E. Y., and Sun, M. — PLDA+: Parallel Latent Dirichlet Allocation with Data Placement and Pipeline Processing, 2011.

[10] Воронцов К.В., Потапенко А.А. — Аддитивная регуляризация тематических моделей, 2014.